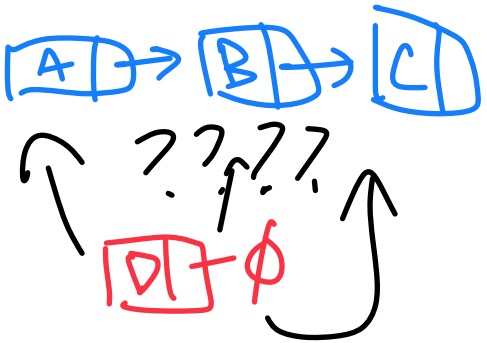
Data Structures

Linked Lists

CS 225

Brad Solomon

September 3, 2025



UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

CS 225 Honors - Course Outline

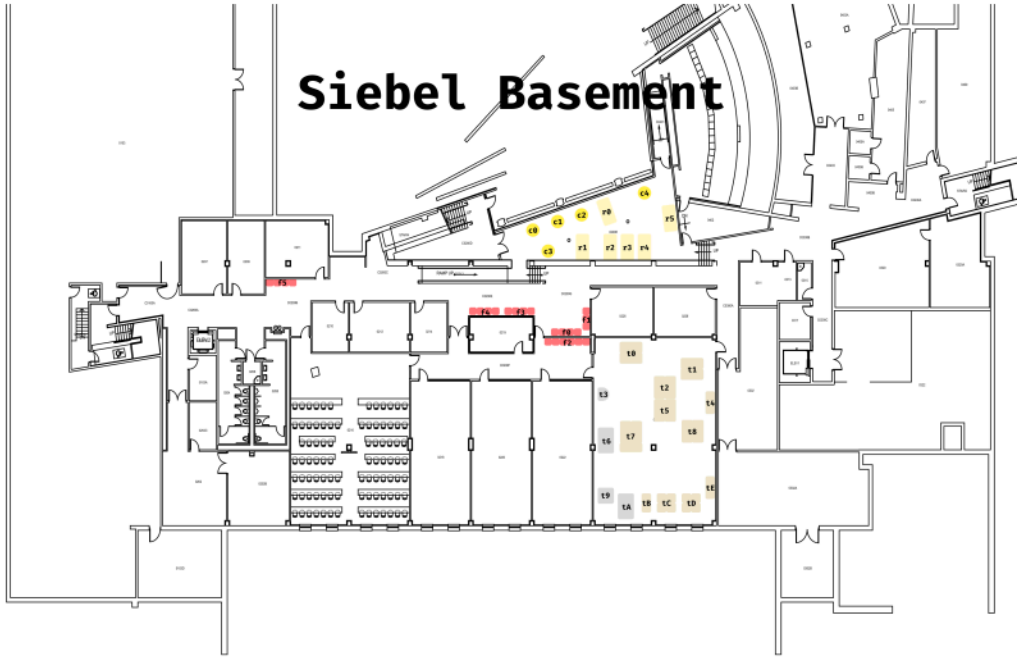
- Lectures every Monday 5pm - 5:50pm @Siebel 0216
- Bi-weekly problem sets (design + analysis of data structures/algorithms)
- Exams : No Additional exams! (This is an independent 0-credit course)
- Pass - Assignment scores total to 50% or more
- Topics - Amortized Analysis, Competitive Analysis, Disjoint-set data structures, Priority queues, Geometric Data structures and Lower bounds.
- Expected Background - CS 173 (Discrete Structures) or equivalent
- Any Questions? - Please contact Harsha Srimath Tirumala (harshast@illinois.edu) Office - Siebel 2322

Office Hour Etiquette

Schedule and link to queue on the website

will be deleted if not followed

Pay attention to the rules!



1. Be in Siebel Basement
2. Tag questions #mp_stickers #TECH
3. Ask **one** specific question
4. Include a specific location
5. Include both your name and Discord ID

MP_Stickers Released

An introduction to assignments in CS 225

A focused look at memory management / Rule of Three

Practice with Image processing / code testing

No extra credit / early submission for this MP!

Be sure to set up your Illinois CS 225 Git repo!

General MP Tips

Start by reading the web page!

Input → output

Take a look at the doxygen for **interface** requirements

The **implementation** is decided by you!

Most autograder tests are provided for you locally

MP_Stickers Highlights

Part 1: test_invert()

Your first example of a templated function!

```
 1  template <class T>
 2  bool test_invert(){
 3    // Your code here
 4    // (As this is the only
 5  templated function, we aren't
 …  using a .hpp file in this
28  instance)
29  }
30
31
32
33
34
```

A Image.h soln

test_invert<T1_Image>();

MP_Stickers Highlights

Part 2: StickerSheet

Big Picture: Memory Management! (Who owns what?)

Read the docs!

◆ render()

Image StickerSheet::render ( ) const

Renders the whole **StickerSheet** on one **Image** and returns that **Image**.

The base picture is drawn first and then each sticker is drawn in order starting with layer zero (0), then layer one (1), and so on.

If a pixel's alpha channel in a sticker is zero (0), no pixel is drawn for that sticker at that pixel. If the alpha channel is non-zero, a pixel is drawn. (Alpha blending is awesome, but not required.)

The returned image always includes the full contents of the picture and all stickers. It should expand in each corresponding direction if stickers go beyond the edge of the picture.

Returns

An **Image** object representing the drawn scene

Exam 0 (9/3 — 9/5)

An introduction to CBTF exam environment / expectations

Quiz on foundational knowledge from all pre-reqs

Practice questions can be found on PL

Topics covered can be found on website

Registration open now

https://courses.engr.illinois.edu/cs225/fa2025/exams/

Learning Objectives

Review fundamentals of linked lists

Implement insert, index, and remove operations

Discuss pointers vs references-to-pointers

List ADT

A list is an **ordered** collection of items

Items can be either **heterogeneous** or **homogenous**

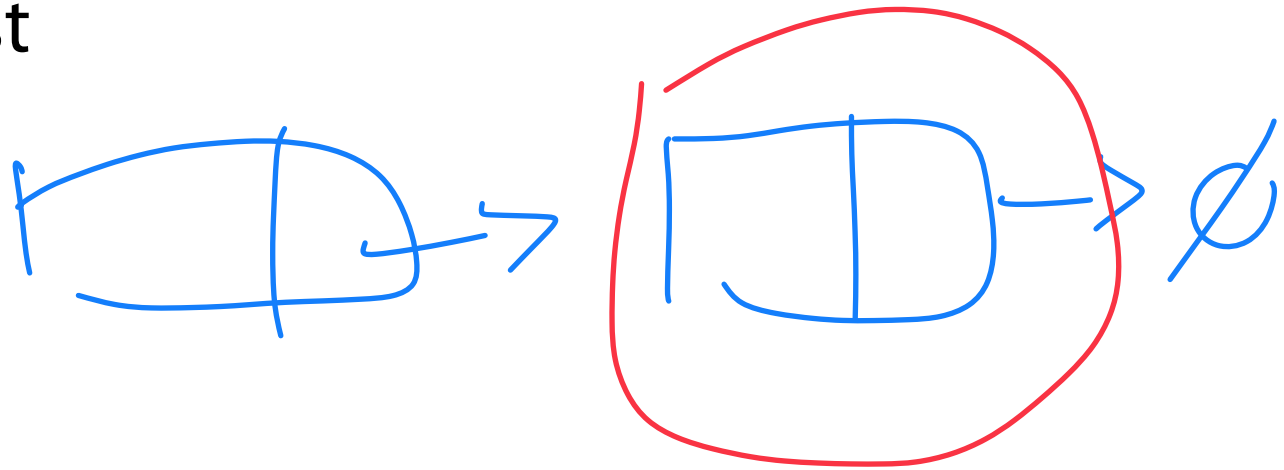
The list can be of a **fixed size** or is **resizable**

A minimal set of operations (that can be used to create all others):

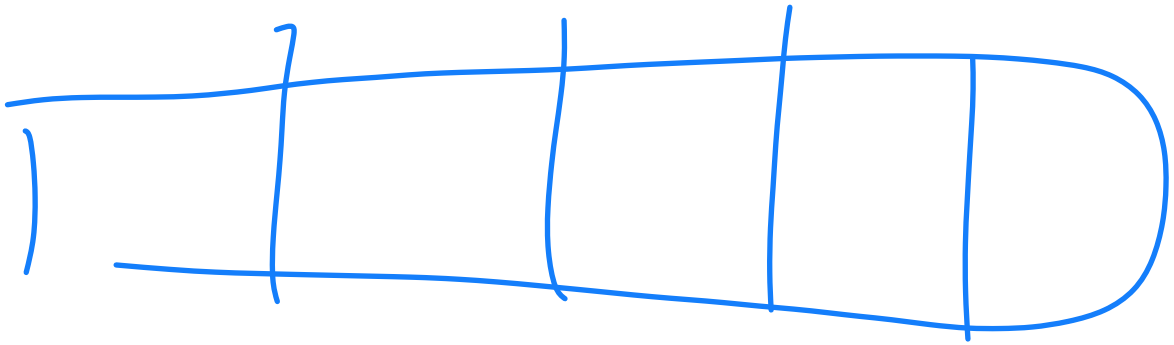
1. Insert
2. Delete
3. isEmpty
4. getData
5. Create an empty list

List Implementations

1. Linked List



2. Array List



List.h

```
 1  template <class T>
 2  class List {
 3    public:
 4      /* ... */
 5    private:
…       class ListNode {
28        T & data;
29        ListNode * next;
30        ListNode(T & data) :
31         data(data), next(NULL) { }
32      };

33      ListNode *head_;
34  };
```

Join Code: 225

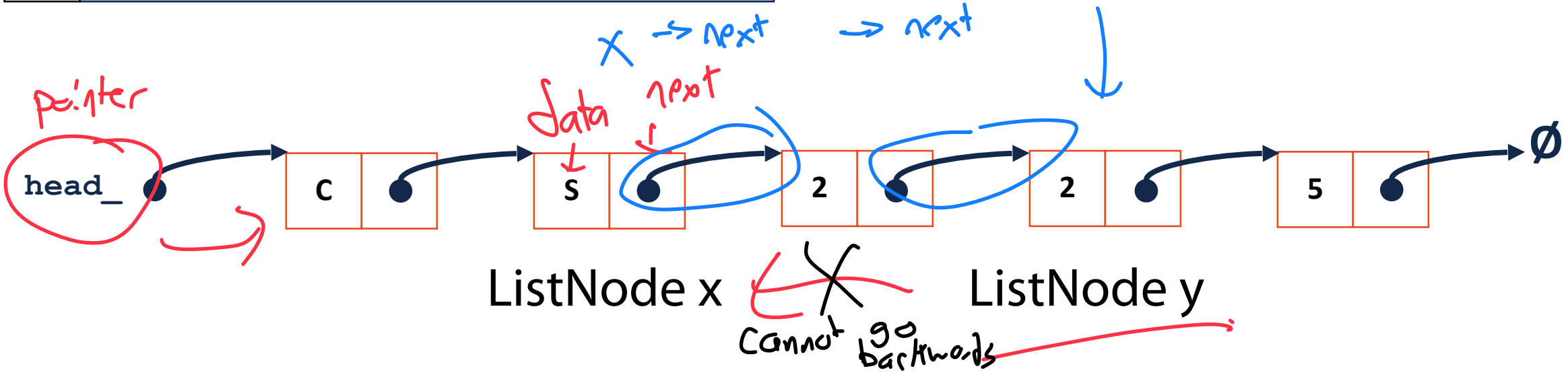
Can we access **x** from **y**?

90% No!

Can we access **y** from **x**?

ListNode *x

x->next->next; // Y



List.h

```
1   #pragma once
2
3   template <typename T>
4   class List {
5     public:
…       /* ... */
28    void insertAtFront(const T& t);
29
30    private:
31      class ListNode {
32        T & data;
33        ListNode * next;
34        ListNode(T & data) :
35         data(data), next(NULL) { }
36      };
37
38      ListNode *head_;
39
40      /* ... */
…
…   };
…
79
```

What is missing in this code?

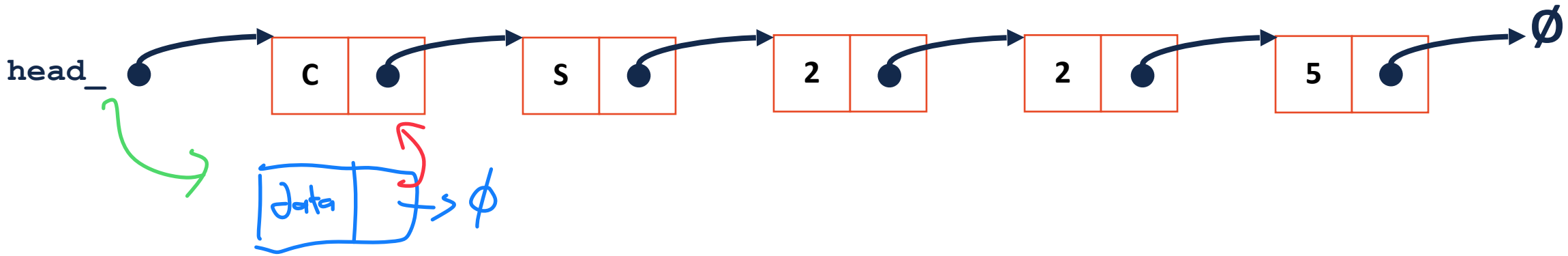
List.h

```
 1  #pragma once
 2
 3  template <typename T>
 4  class List {
 5    public:
 …      /* ... */
28    void insertAtFront(const T& t);
29
30    private:
31      class ListNode {
32        T & data;
33        ListNode * next;
34        ListNode(T & data) :
35         data(data), next(NULL) { }
36      };
37
38      ListNode *head_;
39
40      /* ... */
 …
 …  };
 …
79  #include "List.hpp"
```

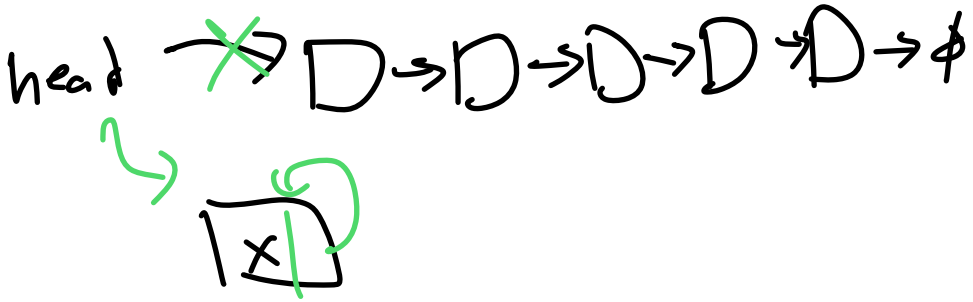
List.hpp

```
 1
 2
 3
 4  void List<T>::insertAtFront(const T& t)
 5  {
 6
 7
 8
 9
10
11
12
13
14
15
16
17
18
19
20
21
22  }
```

Linked List: `insertAtFront(data)`



What if ①, ③, ②?



1 Create ListNode* object X
↳ data = data
↳ next = nullptr

2 Set X->next to be head_;

3 update head_ to point to X

List.h

```
1  #pragma once
2
3  template <typename T>
4  class List {
5    public:
…      /* ... */
28   private:
29     class ListNode {
30       T & data;
31       ListNode * next;
32       ListNode(T & data) :
           data(data), next(NULL) { }
33     };
34
35     ListNode *head_;
36
37     /* ... */
38
39 };
…
…
79 #include "List.hpp"
```

List.hpp

```
1
2
3  template <typename T>
4  void List<T>::insertAtFront(const T& t)
5  {
6
7    ListNode *tmp = new ListNode(t);
8
9
10   tmp->next = head_;
11
12
13   head_ = tmp;
14
15 }
16
17
18
19
20
21
22
```

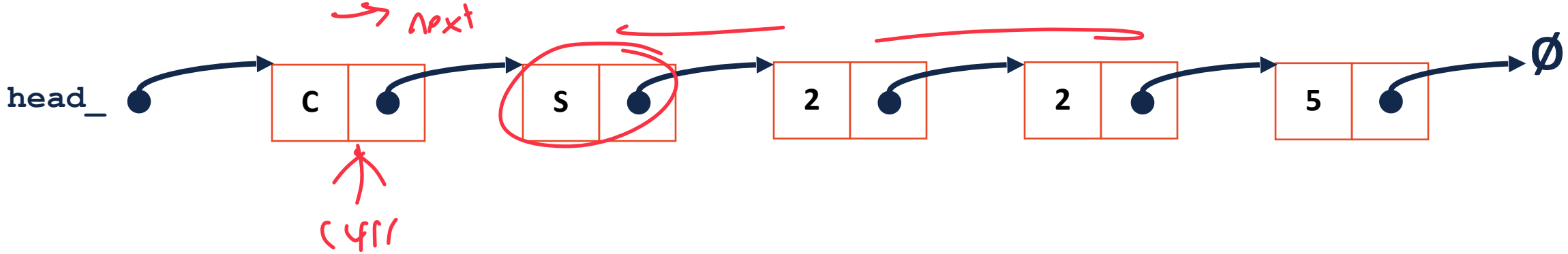
O(1)

O(1)

O(1)

O(1)

Linked List: `insert(data, index)`

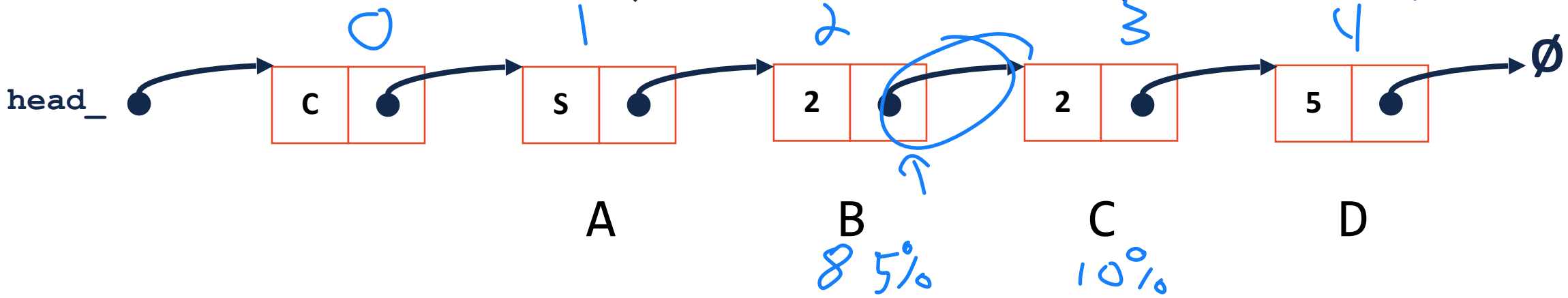



```
ListNode * curr = head;

for (unsigned i = 0;          ; i++){

    curr = curr->next;

}
```

First step is index_

This starter point is Node @ index i-1

Linked List: `insert(data, index)` `insert(d, 3)`



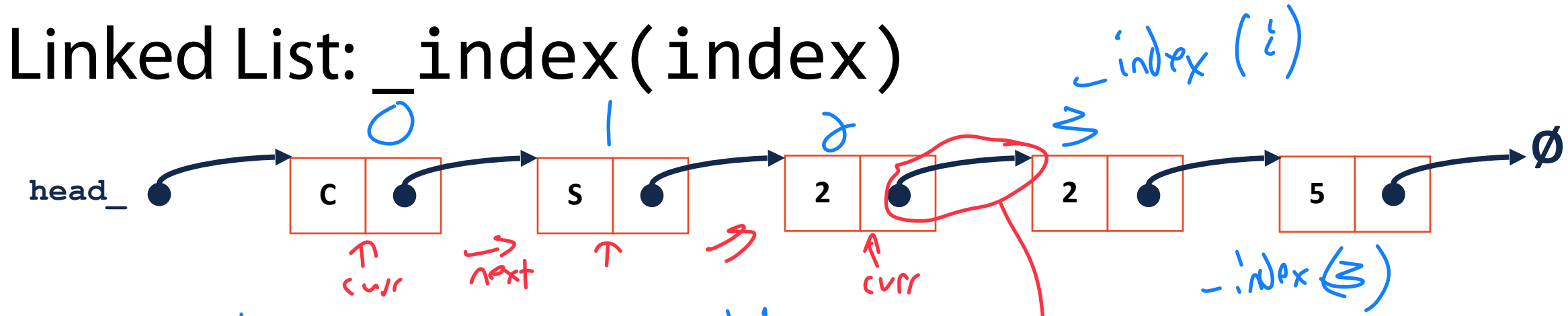
To insert a new ListNode at index 3, we need to **modify** which node?

C → S → 2 → D → 2 → 5

To add at position i, need to modify i-1 position

Join Code: 225

Linked List: _index(index)

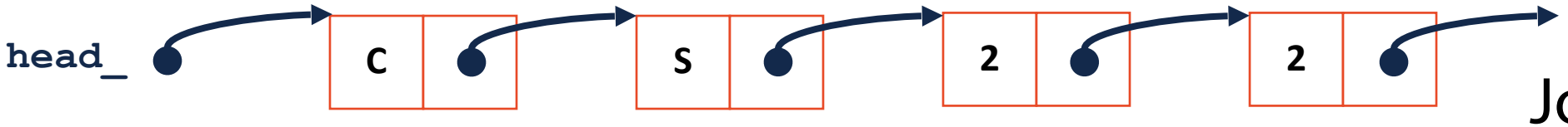


_index returns ref to a pointer

```
List Node * curr;

For (unsigned i = 0; i < index -1 ; i++)

    curr = curr -> next;

}
return curr -> next;
```

arrow in red!

Linked List: _index(index)



Join Code: 225

What should the return type of `_index()` be?

`[template <class T>]`

(A) `T &`

15%

(B) `ListNode`

5%

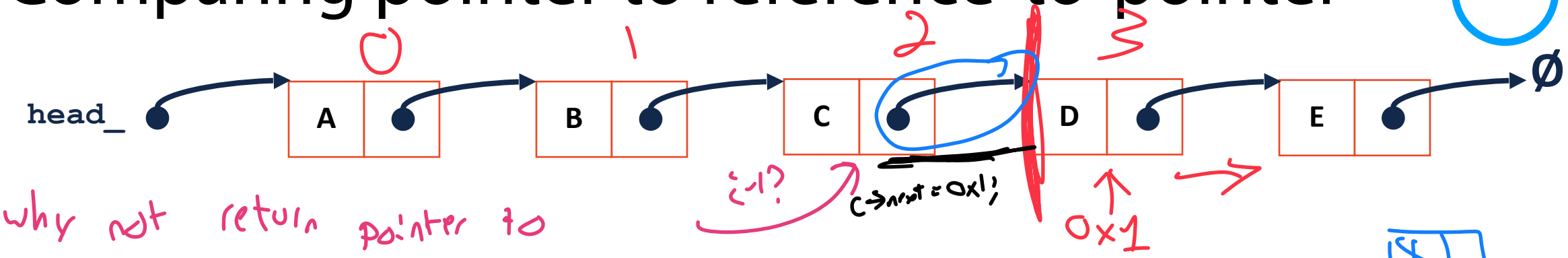
(C) `ListNode *`

65%

(D) `ListNode *&`

20%

Comparing pointer to reference-to-pointer



why not return pointer to

```
ListNode * curr = _index(3);
```

curr has value 0x1

↳ to get data

↳ To get curr->next

```
ListNode *& curr = _index(3);
```

↳ An alias for the variable curr->next

curr->next = 0x3 Access to previous node's next value (only)

All of the above and I can modify the value next at node i-1

A brief tangent…

List.hpp

```
58  template <typename T>
59  typename List<T>::ListNode *& List<T>::_index(unsigned index){
60          return _index(index, head_)
61  }
```

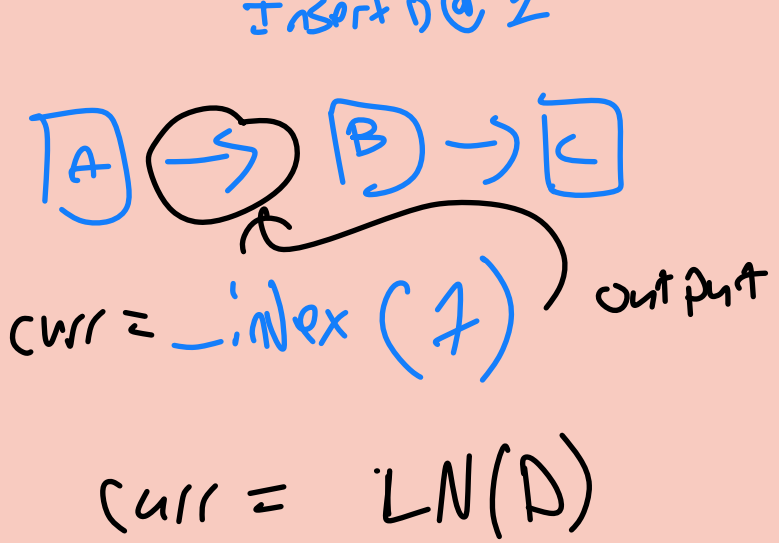
```
63  template <typename T>
64  typename List<T>::ListNode *& List<T>::_index(unsigned index, ListNode *& root){
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80  }
```

A brief tangent...

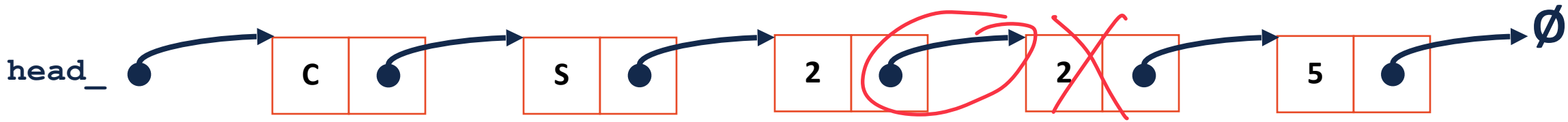
List.hpp

```
58 template <typename T>
59 typename List<T>::ListNode *& List<T>::_index(unsigned index){
60         return _index(index, head_)
61 }
```

```
63 template <typename T>
64 typename List<T>::ListNode *& List<T>::_index(unsigned index, ListNode *& root){
65
66
67
68     if (index == 0){ return root; }
69
70
71
72     if (root == nullptr){ return root; }
73
74
75
76     return _index(index - 1, root -> next);
77
78
79
80 }
```



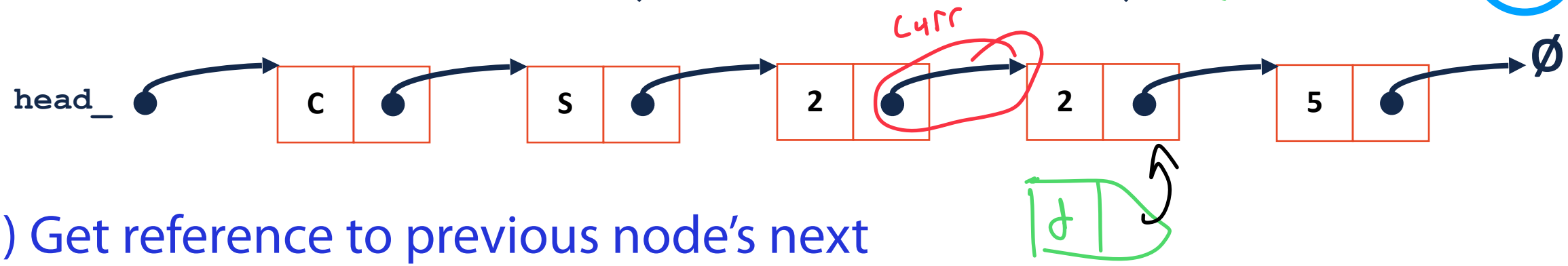
Linked List: `insert(data, index)`



1) Get reference to previous node's next

```
ListNode *& curr = _index(index);
```

Linked List: `insert(data, index)`



1) Get reference to previous node's next

```
ListNode *& curr = _index(index);
```

2) Create new ListNode

```
ListNode * tmp = new ListNode(data);
```

3) Update new ListNode's next

```
tmp->next = curr;
```

4) Modify the previous node to point to new ListNode

```
curr = tmp;
```

insert at Front

* B/c we use ref to pointer in (1) (4) is as easy as declaring a var

Lets compare…

```
1
2  template <typename T>
3  void List<T>::insertAtFront(const T& t)
4  {
5    ListNode *tmp = new ListNode(t);
6
7    tmp->next = head_;
8
9    head_ = tmp;
10
11 }
12
13
14
15
16
17
18
19
20
21
22
```

List.hpp

```
1
2  template <typename T>
3  void List<T>::insert(const T & data,
4  unsigned index) {
5
6
7
8    ListNode *& curr = _index(index);
9
10
11
12   ListNode * tmp = new ListNode(data);
13
14
15
16   tmp->next = curr;
17
18
19
20   curr = tmp;
21 }
22
```

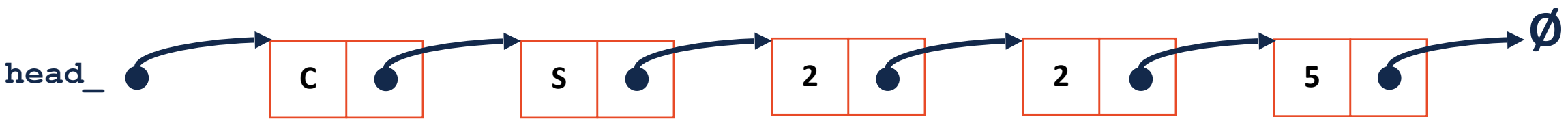
List Random Access [ ]

Given a list L, what operations can we do on L[ ]?

What return type should this function have?

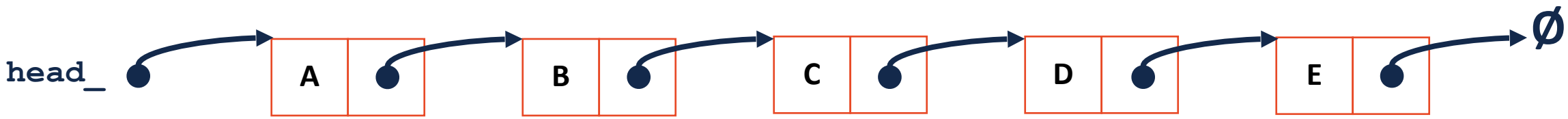
List.hpp

```
48 template <typename T>
49 T & List<T>::operator[](unsigned index) {
50
51
52
53
54
55
56
57
58 }
```

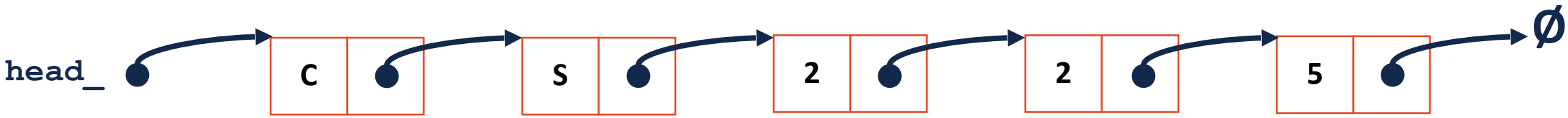


Linked List: `remove(<parameters>)`

What input parameters make sense for remove?

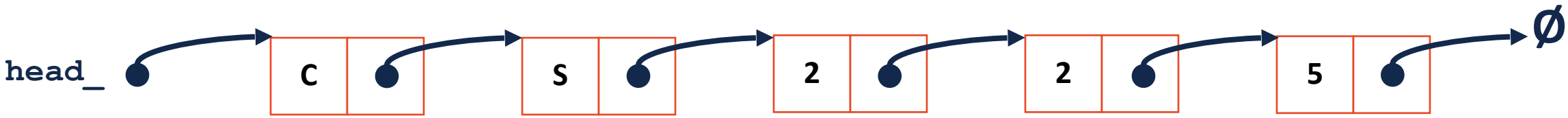


Linked List: `remove(ListNode *& n)`

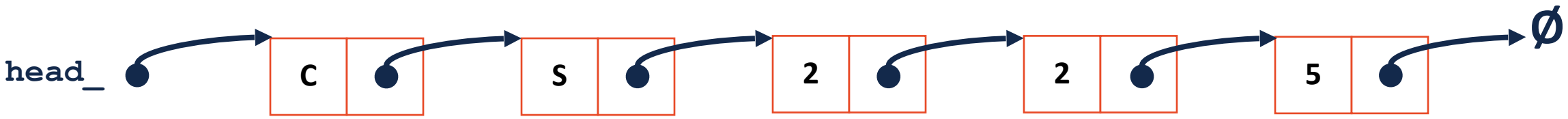


```
103 template <typename T>
104 T List<T>::remove(ListNode *& node) {
105
106
107
108
109
110
111
112 }
```

Linked List: `remove(T & data)`



Linked List: `remove`

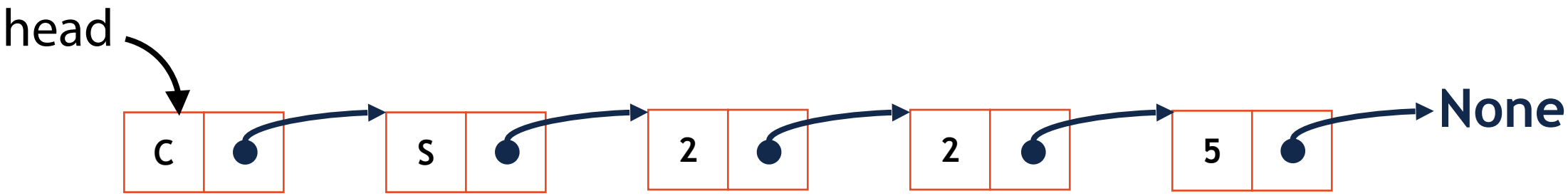


Running time for `remove(ListNode *&)`

Running time for `remove(T & data)`

List Implementations

1. Linked List



2. Array List